Time and Space Optimization of Document Content Classifiers

Dawei Yin, Henry S. Baird and Chang An

Computer Science and Engineering Department
Lehigh University, Bethlehem, PA 18015 USA
day207@lehigh.edu, baird@cse.lehigh.edu, cha305@lehigh.edu

ABSTRACT

Scaling up document-image classifiers to handle an unlimited variety of document and image types poses serious challenges to conventional trainable classifier technologies. Highly versatile classifiers demand representative training sets which can be dauntingly large: in investigating document content extraction systems, we have demonstrated the advantages of employing as many as a billion training samples in approximate k-nearest neighbor (kNN) classifiers sped up using hashed K-d trees. We report here on an algorithm, which we call *online bin-decimation*, for coping with training sets that are too big to fit in main memory, and we show empirically that it is superior to offline pre-decimation, which simply discards a large fraction of the training samples at random before constructing the classifier. The key idea of bin-decimation is to enforce an upper bound approximately on the number of training samples stored in each K-d hash bin; an adaptive statistical technique allows this to be accomplished online and in linear time, while reading the training data exactly once. An experiment on 86.7M training samples reveals a 23-times speedup with less than 0.1% loss of accuracy (compared to pre-decimation); or, for another value of the upper bound, a 60-times speedup with less than 5% loss of accuracy. We also compare it to four other related algorithms.

Keywords: versatile document analysis systems, document content extraction, k Nearest Neighbors, kNN, K-d trees, hashed K-d trees, offline pre-decimation, online bin-decimation

1. INTRODUCTION

We are investigating document image analysis algorithms that work reliably across the broadest possible range of cases, including documents and images that: are written in many languages, typefaces, typesizes, and writing styles; possess a wide range of printing and imaging qualities; obey a myriad of geometric and logical layout conventions; are carefully prepared as well as hastily sketched; may be acquired by geometrically accurate flat-bed scanners, or snapped with hand-held cameras under accidental lighting conditions; are expressed as full color, grey-level, and black-and-white; are of any size or resolution (digitizing spatial sampling rate); and are presented in many image file formats (TIFF, JPEG, PNG, etc), both lossless and lossy. Our previously reported experiments[1,2,3] on document image content extraction have proved the efficacy on such diverse test sets of an approximate k-Nearest Neighbor (kNN) classifier technology sped up with hashed K-d trees .

The training sets necessary for high accuracy are too large to fit in main memory. In previously reported work, we have "randomly decimated" training sets by discarding a large fraction of the training samples at random before constructing the classifier, which we call offline pre-decimation. Here we report an algorithm enhancement, which we call *online bin-decimation*, which continues to allow training with huge training sets, but—as we report here—runs far faster while sacrificing little to no accuracy.

In Section 2, we define our problem precisely. In Section 3, our previously reported hashed K-d tree kNN algorithm is briefly reviewed. Section 4 introduces the bin-decimation algorithm. Section 5 reports and analyzes systematic experiments with bin-decimation. Section 6 concludes with a discussion and an outline of next steps. In the appendix, we briefly compare in contrast four related but inferior algorithms

2. DEFINITIONS

In the document image content extraction applications that we have investigated, we wish to classify each pixel in a set of test document images, to determine which content type—machine-print, hand-written, photo, etc—best characterizes a small region centered on that pixel. Thus each pixel becomes a sample, for training and testing; and each sample is represented by a small number—typically thirty—of numerical features extracted from the region, as described previously.[4]

We have increasingly relied on k-Nearest Neighbor (kNN) classifiers for this purpose, for reasons discussed in.[1,5] Although kNN is often competitive in accuracy with more recently developed methodologies such as neural nets and support-vector machines, it is notoriously slow when implemented by a brute-force algorithm. Another difficulty is that it can be impossible to store all training samples in main memory.

Many techniques for speeding up kNN exist. A popular, practical approach to approximate kNN in higher dimensions is Bentley's k-d trees.[6] Clarkson[7,8] discusses a randomized algorithm for closest-point queries in fixed dimensions that is based on finding Voronoi diagrams of randomly selected subsets of samples. Arya and Mount[9,10] propose that by considering approximate nearest neighbors instead of exact neighbors, they can achieve efficient performance in time and space, regardless of data distribution. Song and Roussopoulos[11] investigate approaches for finding k nearest neighbors for a moving query point.

Indyk et al.[12–15] propose a solution to the approximate nearest neighbor problem using a Locality-Sensitive Hashing scheme. The Approximate Nearest Neighbor Problem is a similarity search problem that tries to find the most similar object to a query object, where objects are defined by a set of features that map to set of points in an attribute space.

To mitigate these problems we have developed approximate kNN algorithms[1–3] based on K-d trees[16] using fixed cuts, also known as multidimensional tries. We briefly review their essential properties. Suppose that for each feature (dimension $i \in \{1, ..., d\}$), lower and upper bounds L_i and U_i on the values of the components x_i are known. Then one may choose to place cuts at midpoints of these ranges, *i.e.* at $(L_i + U_i)/2$, and later, when cutting those partitions, recursively cut at the midpoint of the (now smaller) ranges. A search trie can be constructed in a manner exactly analogous to adaptive K-d trees with the exception that the distribution of the data is ignored in choosing cut thresholds. It will no longer be possible to guarantee balanced cuts and thus most of the time and space optimality properties of adaptive K-d trees are lost. However, there are gains: for example, the values of the cut thresholds can be predicted (they all are of course predetermined by $\{L_i, U_i\}_{i=1,...,d}$). As a consequence, if the total number of cuts r is known, the hyperrectangle within which any query (test) sample x lies can be computed in $\Theta(r)$ time. In practice it can be computed as a side effect of feature extraction and extremely fast, faster than computing a single sample-to-sample distance. We will call these hyperrectangles *bins* (they are sometimes called "cells" or "buckets" in the multi-dimensional data-structure literature).

3. HASHED K-D TREES FOR APPROXIMATE KNN CLASSIFICATION

Bins can be addressed using bit-interleaving, as follows. Let $< d_k, m_k >_{k=\{1,...,r\}}$ be a sequence of cuts, where, at cut k, d_k is the dimension chosen to be cut and m_k is the midpoint of the partition chosen for that cut. Among the partitions of feature space that result, a test sample x will fall in a partition that can be described by a sequence of decisions $b_k = (x_{d_k} > m_k)$ taking on the value False (0) or True (1) as x lies below or above the cut respectively. Equivalently, any bit-sequence $< b_k >_{k=\{1,...,r\}}$ of length r determines the boundaries of some partition, and so can be thought of as an address for it. To summarize: given a test sample x and a sequence of r cuts, we can compute in $\Omega(r)$ time (practically, near-constant time) the bit-interleaved address of the partition within which x lies. By using this address as a hash key, we can exploit hashing techniques to access bins. We call this extremely fast access technique *hashed K-d trees* (for a thorough discussion of their properties see[2]).

Hashed K-d trees support a fast approximation algorithm for kNN: given a test sample, compute its bit-interleaved address, and use it to hash to a bin; then compare the test sample to all the training samples residing in that bin. Resulting errors—results that differ from true kNN—are of two types: *misclassified samples* and *unclassifiable samples*. In the first type, we hash into a bin which contains some training samples, but the result doesn't match the true kNN result: this occurs because not all five true nearest neighbors lie within this bin.

Notation	Meaning
s	A sample.
x_i	The ith feature of a sample.
M	The suggested maximum number of samples in each bin.
$S(b)$	The number of samples which geometrically belong in bin b.
$S_e(b)$	The estimated number of samples which belong in bin b.
$N_t(b)$	At time t, the current number of samples which have fallen into the bin b.
N_t	At time t, the total current number of samples which have been read.
N	Total number of training samples.
$P(s)$	Probability that sample s should be kept.

Table 1. Definitions and notation used in the description of the bin-decimation algorithm.

Errors of the second type occur when the bin contains no training samples at all, and so there is no evidence for any class. (A more detailed discussion can be found in[2]).

It often happens that main memory cannot contain the entire training set. In our previously reported work,[3] we have accommodated to this by *pre-decimation*, randomly discarding a certain fraction of the training samples before constructing the classifier. This speeds up classification by large factors and, perhaps surprisingly, often sacrifices only a little accuracy (as we will show in Section 5). We believe this advantageous tradeoff results from high redundancy in the training data, due to isogeny, that is, the tendency for data in the same image to have been generated by similar processes. In addition to the inevitable loss of accuracy, two problems specific to K-d trees arise:

1. **Bins with too many samples:** the cost of distance computations within the bin may still be unnecessarily high;
2. **Bins with too few samples:** after decimation, some bins may contain no samples at all, causing unclassifiable-sample errors.

4. THE BIN-DECIMATION ALGORITHM

Our model employs all the data structure and algorithms of K-d tree and bit-interleaving addressing. Instead of discarding samples *before* building the K-d tree, we will discard them *while* building the tree, based on the number of samples falling in each K-d bin. The key idea is that the training data is read once and samples are randomly chosen to be discarded based on an on-line adaptive statistical estimation policy. If a particular bin would catch only a few samples—say, fewer than 100—the classifier will keep them all. On the other hand, if a particular bin would catch many samples—say, many more than 100—the classifier will keep only a few. The policy is controlled by an integer run-time parameter, which we call M, suggesting a maximum number of samples to be kept in any bin. Our on-line algorithm for choosing which sampels to keep relies on this assumption:

> ***For every bin, the samples falling in that bin tend to be distributed uniformly within the sequence of training samples, in the order in which they are read.***

If this assumption holds, then, for each sample read, we can use the number of samples in this bin read so far to project the total number of samples in this bin that will be read finally. Please refer to the summary of definitions and notation in Table 1.

When the classifier is reading the samples, the algorithm tries to estimate $S(b)$ by using online statistical information. $S(b)$ is the number of samples which finally fall into the bin b and $S_e(b)$ is the estimate of $S(b)$. According to the assumption, at some moment t, for the bin b the estimated number of samples which will finally fall in bin b is given by

$$S_e(b) = \frac{N_t(b)}{N_t} \times N \tag{1}$$

Each time, after reading a sample s and hashing into its bin b, the algorithm sets $N_t = N_t+1$ and $N_t(b) = N_t(b)+1$ for the next sample. Now, at each time t, we compute $S_e(b)$ for the every bin b. We can then use $S_e(b)$ to calculate the probability that sample s should be kept:

$$P(s) = \frac{M}{S_e(b)} \tag{2}$$

Because $S_e(b) = \frac{N_t(b)}{N_t} \times N$, substitute $S_e(b)$ into the equation, and then we can get

$$P(s) = \frac{M}{\frac{N_t(b)}{N_t} \times N} \tag{3}$$

With this probability, we pseudorandomly keep this sample. The entire *bin-decimation* algorithm is summarized in the Table **Algorithm 1**..

Algorithm 1 Algorithm for bin-decimation

INPUT: training samples set T with the size N , expected number M
OUTPUT: a K-d tree

Initialization $N_t \leftarrow 0$
for all bins b **do**
  $N_t(b) \leftarrow 0$
end for
▷ Sequentially read the samples s from T
for all $s \in T$ **do**
  Hash s into the bin b
  Calculate $P(s)$
  ▷ Update N_t and $N_t(b)$
  $N_t \leftarrow N_t + 1$
  $N_t(b) \leftarrow N_t(b) + 1$
  ▷Generate a random number $r \in (0, 1)$
  $r \leftarrow random(0, 1)$
  if $r < P(s)$ **then**
    load the s into b
  else
    drop s
  end if
end for

This strategy can solve these two problems, (1) bins with too many samples and (2) bins with too few samples, caused by pre-decimation. For the first problem, where a bin b contains much more training samples compared to other bins, it is probably at a given moment t, $N_t(b)$ is relatively larger, so is the estimated number $S_e(b)$. This will result in a much smaller $P(s) = M/S_e(b)$. Accordingly, the samples that fall into this bin will be dropped in a high probability. This dynamical procedure will continue until all the samples are read. In this way, bins only keep a small portion of the original samples. For the second problem, this strategy will hold oppositely effect. If a bin b contains much fewer samples, $N_t(b)$ is much smaller and $P(s)$ is consequently larger. In this case, the samples that fall into this bin will be dropped in a low probability. Therefore, bins will only catch almost all of the original samples.

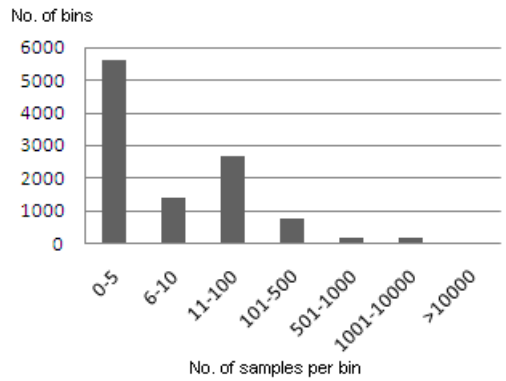

Figure 1. The number of bins as a function of ranges of numbers of training samples per bin. Most bins contain five or fewer samples; and very few contain more than 500 samples.

5. EXPERIMENTS

5.1 Data set

In one small scale experiment, the training set contains 1,658,060 samples and the test set contains 340,054 samples.* Our previous results have shown the success of the kNN classifier with K-d tree. We take this as our baseline method. This baseline method also employs the operation of pre-decimation before loading all the data. In our experiment, we do not compare the brute force kNN classifier, because the approximation method, K-d tree kNN with random decimation, has been proved effective.

5.2 Experimental results

5.2.1 K-d tree kNN without decimation

In this method, we load all the samples into main memory. Figure 1 shows the distribution of the sizes of bins after loading all the training samples (without decimating any samples) and most bins only contain 0-5 samples. Recall that loading all samples into main memory is not practical, because of the memory limit. As we discussed previously, swaping the samples between memory and external storage decreases the efficiency noticeably. The experimental results show that 2245 samples are unclassified, 4.7 billion distances are computed, the accuracy is 78.03% and the CPU runtime is 1554s.

5.2.2 K-d tree kNN with pre-decimation

This method has been used in our previous work[3,4] and has been verified plausible. The idea is straightforward: the K-d Tree kNN classifier is built after randomly decimating some training samples. We tested different ratios of decimation. Because most of the bins only contain 0-5 samples, if we randomly drop samples, bins that contain only 0-5 samples are likely to be emptied. It can be verified in Figure 3 that when the decimation factor is beyond 1/100, the number of unclassified samples increases dramatically. The following Figure 2 shows the runtime and accuracy (on separate scales), as functions of the pre-decimation factor. For example, the pre-decimation factor $r = 1/1000$ means that 999 of 1000 pixels are omitted. In Figure 2, when the pre-decimation factor changes from 1/2 to 1/100, the accuracy drops slightly—6%, but the runtime falls from 1500s to 15s. The result also indicates that the problems existed in both two cases:

1. **Bins with too many samples:** The pre-decimation will not reduce the number of samples effective. That means it may not be decimated into an acceptable range.

*Our test and training images have been collected locally from books, magazines, newpapers, technical articles, and notes of students, etc. All the images are ground-truthed manually using our zoning program. These images along with ground-truth are available for non-commercial research on a request. We use the same feature set as that of our previous work.[3]

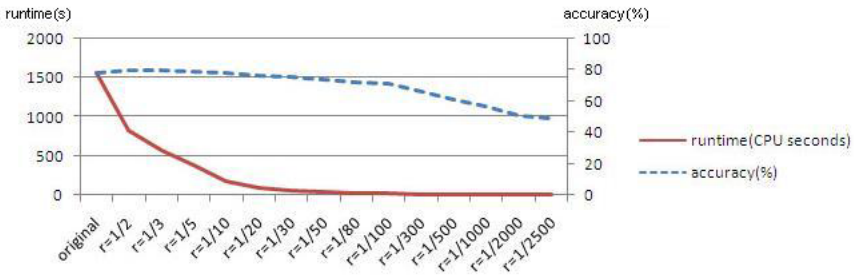

Figure 2. Runtime and accuracy (on separate scales), as functions of the pre-decimation factor. Up to a factor of 1/100, accuracy falls only 6%, while runtime falls by a factor of 100.

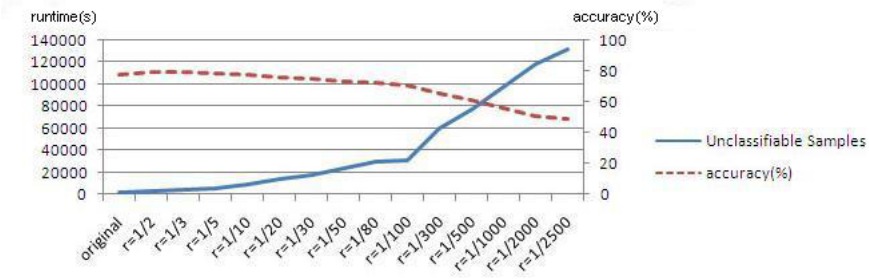

Figure 3. The number of unclassifiable samples, and accuracy, as functions of the pre-decimation factor. For factors beyond 1/100, the number of unclassifiable samples increases dramatically. The same accuracy as in Figure2

2. **Bins with too few samples:** Because of pre-decimation, these bins will lose most samples and become empty bins. In our experiment, especially, dropping beyond 1/100 will cause too many empty bins.

5.2.3 K-d tree kNN with bin-decimation

In this experiment, there is also a runtime parameter, M, which is the suggested number of samples in each bin. To verify that our algorithm can effectively control the number of samples in each bin, we check the distribution of the sizes of bins for different values of M. Figure 4 shows the distribution of sizes of bins, when M=10, 100 and 500 and our strategy can effectively control the size of bin. When M=10, only one bin contains more than 40 samples and none contains more than 50 samples. For M=100 and 500, for the bins with 0-5 samples, comparing to M=10, the numbers of samples in these bins are almost unchanged. These obsevations verify our analysis: if a particular bin catches too many samples, the classifier will drop lot of them;if a particular bin catches only a few samples, the classifier will catch them all, instead of dropping lot of them. Figure 5 shows the runtime and accuracy on different values of M,the runtime parameter controlling maximum bin size. From Figure 5, the runtime drops dramatically from the very beginning while there is almost no lose on the accuracy until M=20. We compare the two methods (pre-decimation and bin-decimation). From Figure 6, the bin-decimation can approach to the 78% faster—18s, and it can achieve 78% which is the same to the original method (without decimating). We still can read the result from the table of details, pre-decimation takes 177s to achieve around 78%.

Because for the case without decimating samples, the result is 78%, if a method can get accuracy-78% in a much less runtime, we think this method can make a good trade between accuracy and runtime. In these cases, for the bin-decimation, we can improve the performance —speedup more than 10 times without losing any accuracy. Further detailed comparison can be found from Table 2 and Table 3. Table 2 shows the results of two methods for around 18s time consuming. From the table, we can see that the accuracy of bin-decimation is 6% larger than pre-decimation. Table 3 shows the results of three methods designed so that thier accuracy are very similar such that we can compare their runtime(for the accuracy $> 77\%$). Comparing to the K-d tree kNN method without decimating, bin-decimation can speedup 100 times without losing any accuracy. Comparing to pre-decimation, it can speedup more than 10 times with no loss of accuracy.

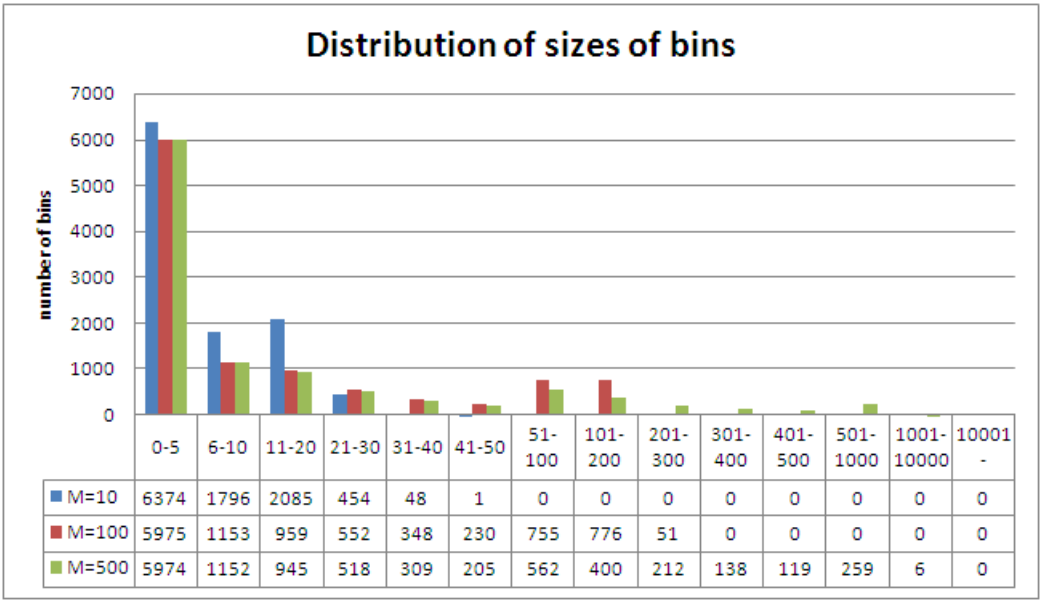

Figure 4. This shows, for three values—10, 100, and 500—of the runtime parameter M, empirical histograms of the number of bins as a function of the number of samples per bin. The data shows that bin-decimation is effective in controlling bin sizes. For example, when M is set to 10, some bins contain more than 10 samples, but not many: one only contains more than 40 samples, and none more than 50.

Classifier	Parameter	Unclassifiable Samples	Distance	Accuracy	Time
pre-decimation	$r = 1/80$	30,335	56,940,476	72.08%	18s
bin-decimation	$M = 30$	2,245	15,151,590	78.02%	18s

Table 2. Comparison of bin-decimation with pre-decimation using parameter choosen so that they consume roughly the same runtime (18 CPU seconds). Note that bin-decimation achieves a higher accuracy (roughly 6% better).

Classifier	Parameter	Unclassifiable Samples	Distance	Accuracy	Time
Without decimation	N/A	2,245	4,739,024,753	78.03%	1554s
pre-decimation	$r = 1/10$	9,521	451,412,941	77.34%	177s
bin-decimation	$M = 20$	2,245	10,585,990	77.1%	15s

Table 3. Comparison of bin-decimation with pre-decimation using parameters choosen so that they acheave roughtly the same accuracy (roughly 77% correct). Note that bin-decimation consumes less runtime (less than 1/10th). It shows the results of three methods designed so that thier accuracy are very similar such that we can compare their runtime.

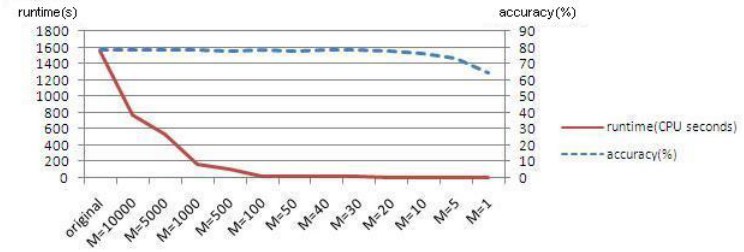

Figure 5. Accuracy and runtime of bin-decimation as functions of M, the runtime parameter controlling maximum bin size. Accuracy remains nearly unaffected until M falls below 5, whereas runtime drops very significantly even for M greater than 100.

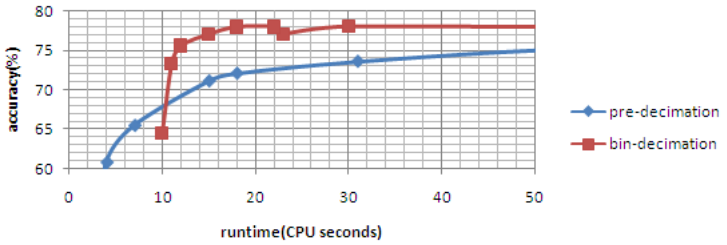

Figure 6. Accuracies of bin-decimation and pre-decimation as functions of runtime comsumed. Bin-decimation is more accurate over a wide range of runtimes, and achieves satisfactory accuracy at far lower runtimes.

Classifier	Parameter	Accuracy	Unclassifiable Samples	Runtime
pre-decimation	$r = 1/8000$	68.12%	161,306	273,829s
bin-decimation	$M = 30$	63.78%	1,164	4,551.31s
bin-decimation	$M = 100$	68.08%	1,164	11,669.54s
bin-decimation	$M = 500$	68.71%	1,164	72,833.57s

Table 4. Results of a large-scale experiment with 86.7M training samples, comparing four methods: pre-decimation (r=1/8000), and bin-decimation for M = 30, 100, and 500. bin-decimation reduced the number of unclassifiable samples by more than two orders of magnitude. Also, bin-decimation (with M=100) is faster by a factor of 23 with only 0.1% loss in accuracy, compared to pre-decimation.

5.2.4 Large scale experiment

In order to verify generality of bin-decimation, a large scale experiment is performed. The training set contains 33 images, a total of 86.7M samples. The test set contains 83 images, a total of 221.6M samples. Experiment environment: high performance computing(Beowulf cluster) at Lehigh University. The HPC Cluster contains 40 nodes and each node is equiped 8 core Intel Xeon 1.8GHz and 16GB memory. From the following Table 4, we can see that bin-decimation can solve the first type of problem, unclassifiable samples. Unlike the pre-decimation—161306 unclassifiable samples, the number of unclassifiable samples is consistently as small as around 1,000. Comparing to the number of training samples 86.7M, it can be ignored. Table 4 shows the results of the large experiment, which are summarized as follow:

A 23-times speedup with less than 0.1% loss of accuracy (for M=100)

A 60-times speedup with less than 5% loss of accuracy (M=30)

It actually improves accuracy (a little: by +0.06%) and still speeds up by a factor of 2.3 (M=500).

6. CONCLUSION AND DISCUSSION

We introduce a new approximate kNN classifier, which achieves better runtime and main memory usage. Our large scale experiment (training sample: 86.7M test samples: 221.6M) shows a 23-times speedup with less than 0.1% loss of accuracy and a 60-times speedup with less than 5% loss of accuracy, compared with pre-decimation. It is a significant improvement of efficiency towards extremely high speed classification on the versatile document image content extraction.

ACKNOWLEDGMENTS

REFERENCES

[1] Casey, M. R. and Baird, H. S., "Towards versatile document analysis systems," in [*Proceedings., 7th IAPR Document Analysis Workshop (DAS'06)*], (February 2006).

[2] Casey, M. R., [*Fast Approximate Nearest Neighbors*], Computer Science & Engineering Dept, Lehigh University, Bethlehem, Pennsylvania (May 2006). M.S. Thesis; PDF available at `www.cse.lehigh.edu/~baird/students.html`.

[3] Baird, H. S., Moll, M. A., and An, C., "Document image content inventories," in [*Proc., SPIE/IS&T Document Recognition & Retrieval XIV Conf.*], (January 2007).

[4] Baird, H. S., Moll, M. A., Nonnemaker, J., Casey, M. R., and Delorenzo, D. L., "Versatile document image content extraction," in [*Proc., SPIE/IS&T Document Recognition & Retrieval XIII Conf.*], (January 2006).
[5] Duda, R. O., Hart, P. E., and Stork, D. G., [*Pattern Classification, 2nd Edition*], Wiley, New York (2001).
[6] Friedman, J. H., Bentley, J. L., and Finkel, R. A., "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software* **3**, 209–226 (September 1977).
[7] Clarkson, K. L., "A randomized algorithm for closest-point queries," *SIAM J. Comput. 17* (1988).
[8] Clarkson, K. L., "An algorithm for approximate closest-point queries," in [*Proceedings of the 10th Annual ACM Symposium on Computational Geometry*], 160–164 (1994).
[9] Arya, S. and Mount, D. M., "Approximate nearest neighbor searching," in [*Proceedings 4th Annual ACM SIAM Symposium on Discrete Algorithms*], 271–280 (1993).
[10] Arya, S., Mount, D. M., Netanyahu, N. S., Silverman, R., and Wu, A. Y., "An optimal algorithm for approximate nearest neighbor searching in fixed dimension," *Journal of the ACM* **45**, 891–923 (November 1998).
[11] Song, Z. and Roussopoulos, N., "K-nearest neighbor search for moving query point," in [*SSTD 2001*], 79–96 (2001).
[12] Datar, M., Immorlica, N., Indyk, P., and Mirrokni, V. S., "Locality-sensitive hashing scheme based on p-stable distributions," in [*Proc., 20th Annual ACM Symposium Computational Geometry*], 253–262, ACM Press (2004).
[13] Datar, M., Immorlica, N., Indyk, P., and Mirrokni, V. S., [*Locality-Sensitive Hashing using Stable Distributions*], ch. 4, MIT Press (2007).
[14] Gionis, A., Indyk, P., and Motwani, R., "Similarity search in high dimensions via hashing," in [*Proc., 25th Int'l Conf. on Very Large Data Bases*], (September 1999).
[15] Indyk, P. and Motwani, R., "Approximate nearest neighbors: Towards removing the curse of dimensionality," in [*Proceedings of the 30th Annual ACM Symposium on Theory of Computing*], 604–613, ACM, New York (1998).
[16] Bentley, J. L., "Multidimensional binary search trees used for associative searching," *Commun. ACM* **18**(9), 509–517 (1975).

APPENDIX

We also experimented with alternative algorithms, but none of them competed successfully with single-pass online algorithm. In all of these algorithms, parameter M is the suggested number of samples in each bin.

Single-pass offline bin-decimation algorithm

The algorithm loads all samples into memory, and then for each bin whose number of samples is larger than M, randomly decimates the samples. This method achieves truly random decimation of a bin to M samples, however, it is often not practical because it must load all samples into memory.

Two-pass offline bin-decimation algorithm

In the first pass, the algorithm reads all samples first but simply counts $S(b)$. In the second pass, because we already know $S(b)$, we use the following probability to randomly decimate the samples: $P(s) = \frac{M}{S_e(b)}$

Figure 7 compares four methods: pre-decimation, single-pass online bin-decimation, single-pass offline bin-decimation and two-pass offline bin-decimation. Single-pass online bin-decimation dominates the other three: it is more accurate over a wide range of runtimes, and achieves satisfactory accuracy at far lower runtimes.

Entropy driven bin-decimation algorithm

We speculated that if some bin is "pure"–that is, contains samples all of which belong to only one class, we think there will be a lower probability of generating classifiable errors and so we can decimate more on this kind of bin. In contrast, if some bin is impure, that is, it contains samples of many classes, we imagine there will be a higher probability of generating classifiable errors and so we should not decimate as much on this kind of bin. To measure impurity, we propose to use entropy.

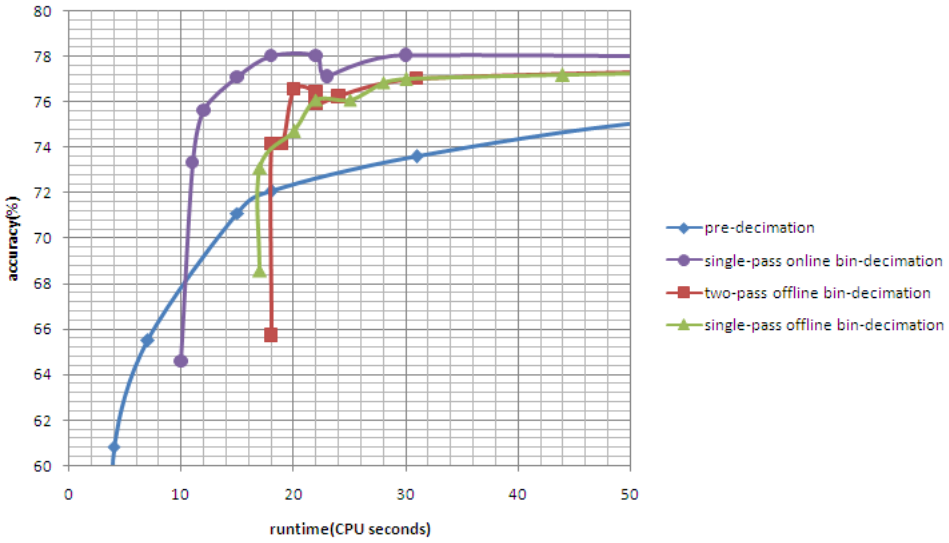

Figure 7. Accuracies of pre-decimation, single-pass online bin-decimation, two-pass offline bin-decimation and single-pass offline bin-decimation as functions of runtime consumed. Single-pass online bin-decimation is more accurate over a wide range of runtimes, and achieves satisfactory accuracy at far lower runtimes.

For each sample x, we know (x_i, c_i), where x_i is the feature vector while c_i is its class. For each bin b, we estimate $P(w_j)$, the probability of the class w_j by computing the fraction of the samples of class w_j in the bin, and estimate impurity, using the standard definition of entropy:

$$i(b) = -\sum_j P(w_j) log_2 P(w_j) \tag{4}$$

Then we exploit impurity in bin-decimation.

$$P(s) = \frac{m + M \times i(b)}{\frac{N_t(b)}{N_t} \times N} \tag{5}$$

Here, two parameters m and M are required, m is a lower bound of the number of the samples in the bins. In the range $[m, M]$, the number of samples is adjusted by impurity. If $S(b) > M$ and the bin is impure, we decimate to M. If however the bin is pure, we decimate more, leaving fewer than M samples. We tested the method on a small data set and got a result which is not as good as we expected: using m = 50 and M = 500, we observe the result accuracy = 0.760 and runtime = 49.8s. Comparing single-pass online in-decimation on the same data, using M = 50, the result is accuracy = 0.777 and runtime = 12.8s. We had hoped that because m also equals 50, the entropy-driven method might be better than bin-decimation. However, we found that the entropy-driven method is worse in both run runtime and accuracy.

Soft single-pass online bin-decimation algorithm

The other algorithms can be called "hard decimation" algorithms, because if $S(b) > M$, the decimation is applied so that the number of the samples remaining is approximatly M.

There is an alternative, which we call "soft decimation", where we allow the number of samples to exceed M, but to grow more slowly than when the number of samples per bin is less than M. For example, if $S(b) > M$, we may keep $M + log(S(b) - M)$ samples.

We use the following equation to calculate the probability online:

$$P(s) = \frac{M + log(\frac{N_t(b)}{N_t} \times N - M)}{\frac{N_t(b)}{N_t} \times N} \tag{6}$$

We also test this method on a small data set and the result is also not as good as we expected. For soft bin-decimation, we uses M = 50, and then we observe accuracy = 0.779 and runtime = 13.5s. Comparing single-pass online bin-decimation on the same data, using M = 50, the result was accuracy = 0.770 and runtime = 12.6s. So there is a slight improvement in accuracy because of more samples in some bins. We think it is not a significant improvement and we also notice the increase of runtime.